Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis in Computer Science
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023, 17 pages
   https://www.sciencedirect.com/science/article/pii/S0167404823002067

2. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
   Under review, 17 pages
   https://arxiv.org/pdf/2309.07638.pdf

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Moving Target Defense (MTD 2022), 12 pages
   https://dl.acm.org/doi/abs/10.1145/3560828.3564007

4. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
   https://doi.org/10.14722/madweb.2021.23004

5. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs, 4 pages
   https://doi.org/10.1145/3397537.3397567

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019), 10 pages
   https://doi.org/10.1145/3358504.3361228

ACKNOWLEDGEMENT

Contents

Part I

Thesis

1 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

The company Polyverse[1] applies randomization on a commercial scale. They provide each client with a unique Linux distribution compilation by scrambling the Linux packages at the source code level.

1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

[0] Comp. time 2023/10/17 16:29:53

[1] `https://polyverse.com/`

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

2
BACKGROUND AND STATE OF THE ART

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss WebAssembly, focusing on its design and security model. Besides, we discuss the current state-of-the-art in the area of WebAssembly program analysis. In Section 2.2 we discuss related works in the area of Software Diversification. Moreover, we delve into the open challenges regarding the diversification of WebAssembly programs.

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017 and extended by Rossberg et al. in 2018 [**?**]. It is designed to be fast, portable, self-contained, and secure.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup time and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecode Alliance proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], i.e., IoT and Edge computing [**?** **?**].

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [**?**]. Other applications include text processing, visualization, media processing, programming language testing, online gambling,

[0]Comp. time 2023/10/17 16:29:53

bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly suitable for Function as a Service (FaaS) platforms like Cloudflare and Fastly. The broad spectrum of applicability and the rapid adoption of WebAssembly have resulted in demands for additional features. However, not all these demands align with its original specification. Thus, since the introduction of WebAssembly, various extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. To become a standard, a proposal must fulfill certain criteria, including having a formal specification and at least two independent implementations, e.g., two different engines. Notably, even after adoption, new extensions are optional; the core WebAssembly remains untouched and continues to be referred to as version 1.0. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed are applicable to future WebAssembly versions.

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

```
1  // Some raw data
2  const int A[250];
3
4  // Imported function
5  int ftoi(float a);
6
7  int main() {
8      for(int i = 0; i < 250; i++) {
9          if (A[i] > 100)
10             return A[i] + ftoi(12.54);
11     }
12
13     return A[0];
14 }
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

In Listing 2.1 and Listing 2.2, we present a C program alongside its corresponding WebAssembly binary. The C function encompasses various elements such as heap allocation, external function usage, and a function

definition that includes a loop, conditional branching, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is displayed in its textual format, known as WAT[1]. The static memory declared in line 2 of Listing 2.1 is allocated within the Wasm binary's linear memory, as illustrated in line 47 of Listing 2.2. The function prototype in line 5 of Listing 2.1 is converted into an imported function, as seen in line 8 of Listing 2.2. The main function, spanning lines 7 to 14 in Listing 2.1, is transcribed into a Wasm function covering lines 12 to 38 in Listing 2.2. Within this function, the translation of various C language constructs into Wasm can be observed. For instance, the `for` loop found in line 8 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The loop's breaking condition is converted into a conditional branch, as shown in line 25 of Listing 2.2.

There exist several compilers that turn source code into WebAssembly binaries. For example, LLVM compiles to WebAssembly as a backend option since its 7.1.0 release[2], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig [**?**] reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a modular and state-of-the-art optimization infrastructure for WebAssembly binaries. In parallel, the KMM framework[4] has incorporated WebAssembly as a compilation target.

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[5] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[6] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, packaging the interpreter and the code in one single standalone WebAssembly binary is still immature and faces challenges. For example, the absence of JIT compilation for the "interpreted" code makes it less suitable for long-running tasks [**?**]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

[1] The WAT text format is primarily designed for human readability and for low-level manual editing.

[2] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

[3] A subset of the TypeScript language

[4] `https://kotlinlang.org/docs/wasm-overview.html`

[5] `https://github.com/bytecodealliance/javy`

[6] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

```
1  ; WebAssembly magic bytes(\0asm) and version (1.0) ;
2  (module
3  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
4    (type (;0;) (func (param f32) (result i32)))
5    (type (;1;) (func))
6    (type (;2;) (func (result i32)))
7  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
8    (import "env" "ftoi" (func $ftoi (type 0)))
9  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
10   (@custom "name" "...")
11 ; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
12   (func $main (type 2) (result i32)
13     (local i32 i32)
14     i32.const -1000
15     local.set 0
16     block  ;label = @1;
17       loop  ;label = @2;
18         i32.const 0
19         local.get 0
20         i32.add
21         i32.load
22         local.tee 1
23         i32.const 101
24         i32.ge_s
25         br_if 1 ;@1;
26         local.get 0
27         i32.const 4
28         i32.add
29         local.tee 0
30         br_if 0 ;@2;
31       end
32       i32.const 0
33       return
34     end
35     f32.const 0x1.9147aep+3
36     call $ftoi
37     local.get 1
38     i32.add)
39 ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
40   (memory (;0;) 1)
41 ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
42   (global (;4;) i32 (i32.const 1000))
43 ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
44   (export "memory" (memory 0))
45   (export "A" (global 2))
46 ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
47   (data $data (0) "\00\00\00\00...")
48 ; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
49   (@custom "producers" "...")
50 )
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

2.1.2 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-

byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A Wasm binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty. For clarity, we also annotate sections as comments in the Wasm code in Listing 2.2.

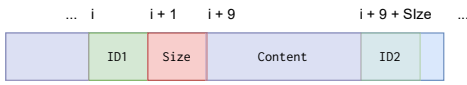


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

A Wasm binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to expedite the compilation process either through parallel parsing or by disregarding *Custom Sections*. Moreover, the *Code Section*'s instructions are further compacted through the use of the LEB128[7] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

2.1.3 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of Wasm programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control-flow and the analysis of WebAssembly binaries.

Execution Stack: At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Section 2.1.2. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instruction

[7] `https://en.wikipedia.org/wiki/LEB128`

starts and loop starts. Control frames manage function calls and function returns. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

Functions: At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, or modifies the control-flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 12 and Line 38 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. Notice that, numeric instructions are annotated with its corresponding type. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` in line 38. As the listing also shows, instructions are annotated with a numeric type.

Memory model: A WebAssembly module instance incorporates three key types of memory-related components: linear memory, local variables and global variables. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [**?**]. Managed refers to components that are exclusively modified by the host engine at the lowest level, e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include multiple linear memory instances, which are contiguous arrays of bytes. These are accessed using 32-bit integers (`i32`) and are shareable only between the initiating engine and the WebAssembly binary. Generally, these linear memories are considered to be unmanaged, e.g., line 21 of Listing 2.2 shows an explicit memory access opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 42 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance. They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

WebAssembly module execution: While a WebAssembly binary could be interpreted, the most practical approach is to JIT compile it into machine

code. The main reason is that WebAssembly is optimized and closely aligned with machine code, leading to swift JIT compilation for execution. Browser engines such as V8[8] and SpiderMonkey[9] utilize this strategy when executing WebAssembly binaries in browser clients. Once JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment exclusively through imported functions. The communication between the host and the WebAssembly module execution is typically facilitated by trampolines in the JITed machine code.

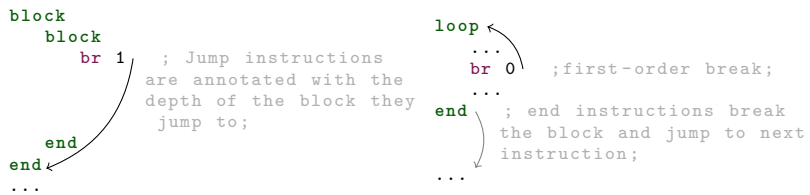
WebAssembly standalone engines: While initially intended for browsers, WebAssembly has undergone significant evolution, primarily due to WASI[**?**]. WASI establishes a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate Wasm binaries that implement WASI, which allows execution in standalone engines. These binaries can then be executed by standalone engines across a variety of environments, including the cloud, servers, and IoT devices [**?**]. Similarly to browsers, these engines often translate WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process. Standalone engines such as WASM3[10], Wasmer[11], Wasmtime[12], WAVM[13], and Sledge[**?**] have been developed to support both WebAssembly and WASI. In a related development, Singh et al.[**?**] have created a WebAssembly virtual machine specifically designed for Arduino-based devices.

2.1.4 WebAssembly's control-flow

A WebAssembly function groups instructions into blocks, with the function's entrypoint acting as the root block. In contrast to conventional assembly code, control-flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is simply the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they commence and conclude. By design, a block cannot reference or execute code from external blocks.

[8] `https://chromium.googlesource.com/v8/v8.git`
[9] `https://spidermonkey.dev/`
[10] `https://github.com/wasm3/wasm3`
[11] `https://wasmer.io/`
[12] `https://github.com/bytecodealliance/wasmtime`
[13] `https://github.com/WAVM/WAVM`

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 34 of Listing 2.2, the execution would proceed to line 35. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 30 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design hardens the rewriting of Wasm binaries. For instance, if an outer block is removed, the depth of the break instructions within nested blocks must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control-flow graph to determine the enclosing block depth for each break instruction.

2.1.5 Security and Reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need. This paper presents a review of the most relevant tools in this field, focusing on those capable of providing security guarantees for WebAssembly binaries.

Static analysis: SecWasm[**?**] uses information control-flow strategies to identify vulnerabilities in WebAssembly binaries. Conversely, Wasmati[**?**] employs code property graphs for this purpose. Wasp[**?**] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. VeriWasm[**?**], an offline verifier designed specifically for native x86-64 binaries JITed from WebAssembly, adopts a unique approach. While these tools emphasize specific strategies, others adopt a more holistic approach. CT-Wasm[**?**

], verifies the implementation of cryptographic algorithms in WebAssembly. Similarly, Vivienne applies relational Symbolic Execution (SE) to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations[**?**]. For example, both Wassail[**?**] and WasmA[**?**] provide a comprehensive static analysis framework for WebAssembly binaries. However, static analysis tools may have limitations. For instance, a newly, semantically equivalent WebAssembly binary may be generated from the same source code bypassing or breaking the static analysis [**?**]. If the WebAssembly input differs from the input used during sound analysis [**?**], the vulnerability may go unnoticed. Thus, there may be a lack of subjects to evaluate the effectiveness of these tools.

Dynamic analysis: Dynamic analysis involves tools such as TaintAssembly[**?**], which conducts taint analysis on WebAssembly binaries. Fuzzm[**?**] identifies vulnerabilities in host engines by conducting property fuzzing through WebAssembly binary execution. Furthermore, Stiévenart and colleagues have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. This technique aids in debugging, understanding programs, and conducting security analysis. However, Wasabi[**?**] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. Similar to static analysis, these tools typically analyze software behavior during execution, making them inherently reactive. In other words, they can only identify vulnerabilities or performance issues while pseudo-executing input Wasm programs. Thus, facing an important limitation on overhead for real-world scenarios.

Protecting WebAssembly binaries and runtimes: The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and fortifying the security of the entire execution runtimes ecosystem. For instance, Swivel[**?**] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries, particularly in Function-as-a-Service (FaaS) platforms such as Fastly. Conversely, WaVe[**?**] introduces a mechanized engine for WebAssembly that facilitates differential testing. This engine can be employed to detect anomalies in engines running Wasm-WASI programs. Much like static and dynamic analysis tools, these tools may suffer from a lack of WebAssembly inputs, which could affect the measurement of their effectiveness.

WebAssembly malware: Since the introduction of Wasm, the Web has consistently experienced an increase in cryptomalware. This rise primarily stems from the shift of mining algorithms from CPUs to Wasm, a transition driven by notable performance benefits [**?**]. Tools such as MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] employ static analysis with machine learning

techniques to detect browser-based cryptomalwares. Conversely, SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**] leverage dynamic analysis techniques to achieve a similar objective. VirusTotal[14], a tool incorporating over 60 commercial antivirus systems as black-boxes, is capable of detecting cryptomalware in Wasm binaries. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bahnsali et al. seminal work[**?**] demonstrate that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

2.1.6 Open challenges

Despite progress in WebAssembly analysis, numerous challenges remain. WebAssembly, though deterministic and well-typed by design, boasts an emerging ecosystem susceptible to a variety of security threats. First, most existing WebAssembly research is reactive, focusing on detecting and fixing vulnerabilities already reported. This approach leaves WebAssembly binaries and runtime implementations potentially open to unidentified attacks. Second, side-channel attacks present a significant risk. Genkin et al., for example, illustrated how WebAssembly could be manipulated to extract data via cache timing-side channels [**?**]. Furthermore, research conducted by Maisuradze and Rossow demonstrated the potential for speculative execution attacks on WebAssembly binaries [**?**]. Rokicki et al. disclosed the possibility for port contention side-channel attacks on WebAssembly binaries in browsers [**?**]. Finally, the binaries themselves may be inherently vulnerable. For example, studies by Lehmann et al. and Stiévenart et al. suggested that flaws in C/C++ source code could infiltrate WebAssembly binaries [**?** **?**].

This dissertation presents toolsets, approaches and methodologies designed to enhance WebAssembly security proactively through Software Diversification. First, Software Diversification could expand the capabilities of the mentioned tools by incorporating diversified program variants, making it more challenging for attackers to exploit any missed vulnerabilities. Generated as proactive security, these diversified variants can simulate a broader set of real-world conditions, thereby making WebAssembly analysis tools more accurate. Second, we noted that current solutions to mitigate side-channel attacks on WebAssembly binaries are either specific to certain attacks or need the modification of runtimes, e.g., Swivel as a cloud-deployed compiler. Software Diversification could mitigate yet-unknown vulnerabilities on WebAssembly binaries by generating diversified variants in a platform-agnostic manner.

[14] `https://www.virustotal.com`

2.2 Software diversification

Software Diversification has been extensively studied in recent decades. This section explores its current state of the art. Software diversification involves the synthesis, reuse, distribution, and execution of different, functionally equivalent programs. As outlined in Baudry et al.'s survey [**?**], software diversification falls into five usage categories: reusability [**?**], software testing [**?**], performance [**?**], fault tolerance [**?**], and security [**?**]. Our work specifically contributes to the last two categories. This section presents related works, emphasizing how they generate diversification and apply it to WebAssembly.

2.2.1 Generation of Software Variants

Software variants are functionally equivalent versions of an original program, created through software diversification at different stages of the software lifecycle, such as the source code or machine code levels. The diversification can be either natural [**?**], arising spontaneously but requiring significant human effort, or artificial [**?**], which is automated and the focus of our work in the context of WebAssembly. The concept of software variants dates back to Randell's 1975 work [**?**], which introduced the idea of fault-tolerant instruction blocks. Later, artificial software diversification was further developed through rewriting strategies, as proposed by Cohen and similarly by Forrest in the 1990s [**?** **?**]. These strategies consist of rule sets for altering software components to create functionally equivalent but distinct programs. Our work builds on these foundational studies and focuses on artificial diversification techniques, particularly for WebAssembly, drawing insights from significant works by Baudry et al. [**?**] and Jackson et al. [**?**]. In the following, we group the major strategies used to artificially generate software variants and their current state wrt WebAssembly.

Replacement of Equivalent Instructions: One can replace sections of programs with semantically equivalent code. This method requires substituting the original code with identical arithmetic expressions or injecting instructions that do not alter the final computation outcome. There are primarily two methods for generating such equivalent code: rewriting rules and enumerative synthesis. In the first method, manual rewriting rules dictate the replacement strategies. A rewriting rule consists of a code segment and its semantically identical substitution. For instance, Cleemput et al. [**?**] and Homescu et al. [**?**] introduce NOP instructions to produce statically varied versions. In these studies, the rewriting rule is expressed as `instr => (nop instr)`, implying a `nop` operation followed by the instruction as a valid substitute. In contrast, enumerative synthesis explores all potential programs specific to a language. In this domain, Jacob et al. [**?**] introduced a technique called superdiversification

for x86 binaries. Similarly, Tsoupidi et al. [**?**] presented Diversity by Construction, a constraint-based compiler that creates software diversity for the MIPS32 architecture. Their technique employs a constraint solver to generate program variants that are semantically equivalent by design. Compared to other methods, Jacob et al. and Tsoupidi et al.'s work does not need manually written replacement strategies, but their reach is limited by theorem solvers. While their techniques can be implemented in any language, they are not directly applicable to WebAssembly. For instance, while the studies of Cleemput et al. and Homescu et al. are directly applicable to WebAssembly, since WebAssembly typically compiles later, this specific strategy could fall into a *non preserved* category, meaning JIT compilers could eliminate this diversification strategy by simply applying straightforward optimizations. Conversely, the application of enumerative synthesis to WebAssembly has not been explored, and it is actually one of our contributions. As further stated in Chapter 3, implementing enumerative synthesis for WebAssembly poses language-specific challenges.

Instruction Reordering: This strategy involves reordering independent instructions or entire program blocks. The location of variable declarations may also change if compilers reorder them in the symbol tables. This prevents static examination and analysis of parameters and alters memory locations. In this area, Bhatkar et al. [**? ?**] proposed the random permutation of variable and routine order for ELF binaries. Such strategies are not implemented for WebAssembly to the best of our knowledge. Yet, it is part of our technical contributions.

Adding, Changing, Removing Jumps and Calls: This strategy generates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] primarily illustrated this concept by inserting random jumps in programs. Pettis and Hansen [**?**] suggested splitting basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inlined basic blocks of code as an LLVM pass. In their approach, each de-inlined code transforms into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**], replacing function calls with indirect pointer calls in C source code, allowing post-binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm functions. As previously discussed in Section 2.1.4, the control flow of WebAssembly is restricted, making them impractical to directly port these strategies to WebAssembly. Yet, we implement a similar strategy in our diversification tools for WebAssembly, as discussed in Chapter 3.

Program Memory and Stack Randomization: This strategy alters the layout of programs in the host memory. Additionally, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**? ?**] proposes

to randomize the base addresses of applications and library memory regions in ELF binaries. Tadesse Aga and Autin [**?**], and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [**?**] suggest separating a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization. This makes it very challenging for an attacker to ignore the key to inject executable code. This strategy disrupts the predictability of program execution and mitigates certain exploits such as speculative execution. We have not found any work that applies this strategy to WebAssembly. As part of the evaluation of our contributions in Chapter 4, we demonstrate that our tools could inherently affect the memory layout of WebAssembly programs. Thus, we consider this strategy as part of our contributions.

ISA Randomization and Simulation This strategy involves using a key to cypher the original program binary into another encoded binary. Once encoded, the program can only be decoded at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**], and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client. Yet, both strategies aim to tackle program analysis from potential attackers. Moreover, this strategy faces a performance penalty, specially for WebAssembly, due to the decoding process as shown in WASMixer evaluation [**?**].

Equivalence checking between program variants is a vital component for any program transformation task, ranging from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. It proves that two pieces of code or programs are functionally equivalent [**?**]. Cohen [**?**] simplifies the checking process with the following property: two programs are equivalent if, given identical inputs, they produce identical outputs. We adopt this definition of *functional equivalence* throughout this dissertation. In Software Diversification, equivalence checking seeks to preserve the original functionality of programs while varying observable behaviors. Two programs, for instance, can differ statically and still compute the same result. We outline two methods to

check variant equivalence: by construction and automated equivalence checking. In Chapter 3, we discuss the primary advantages and limitations in practice for both approaches within the scope of our contributions.

Equivalence checking by construction: The equivalence property is often guaranteed by construction. Cleemput et al. [**?**] and Homescu et al. [**?**], for example, design their transformation strategies to generate semantically equivalent program variants. However, developer errors can occur in this process, necessitating further validation. The test suite of the original program can serve as a check for the variant. If the program variant passes the test suite [**?**], it can be considered equivalent to the original. However, this technique is limited by the need for a preexisting test suite. Should the test suite not exist, an alternative technique is required for equivalence checking.

Automated equivalence checking: In the absence of a test suite or a technique that inherently implements the equivalence property, the works mentioned earlier use theorem solvers (SMT solvers) [**?**] to prove equivalence. The central idea for SMT solvers is to convert the two code variants into mathematical formulas. The SMT solver then checks for counter-examples. When it finds a counter-example, there is an input for which the two mathematical formulas yield different outputs. The primary limitation of this technique is that not all algorithms can be translated into a mathematical formula, such as loops. Nevertheless, this technique is frequently used for checking no-branching-programs like basic block and peephole replacements [**?**]. An alternative method for checking program equivalence, similar to using SMT solvers, involves the use of fuzzers [**?**]. Fuzzers randomly generate inputs that yield different observable behaviors. If two inputs produce a different output in the variant, the variant and the original program are not equivalent. The primary limitation for fuzzers is that the process is notably time-consuming and necessitates manual introduction of oracles.

2.2.2 Variants deployment

Program variants, once generated and verified, may be utilized in two primary scenarios: Randomization or Multivariant Execution (MVE) [**?**]. Additionally, these variants serve both defensive and offensive purposes [**?**].

Randomization: In the context of our work, the term *Randomization* denotes a program's ability to present different variants to different clients. In this setup, a program, chosen from a collection of variants (referred to as the program's variant pool), is assigned to a random client during each deployment. Jackson et al. [**?**] define the variant pool in Randomization as herd immunity, as vulnerable binaries can only affect a segment of the client community. El-Khalil and colleagues [**?**] suggest employing a custom compiler to generate varying binaries from the

compilation process. They adapt a version of GCC 4.1 to partition a conventional stack into several component parts, termed multistacks. Similarly, Singhal and colleagues, propose Cornucopia [**?**]. Cornucopia generates multiple variants of a program by using different compiler flag combinations. Aga and colleagues [**?**], contributing to this discussion, propose the generation of program variants through the randomization of its data layout in memory. This method allows each variant to operate on the same data in memory but at different memory offsets. Randomization can also be applied to virtual machines and operating systems. On this note, Kc et al. [**?**] establish a unique mapping between artificial CPU instructions and actual ones, enabling the assignment of various variants to specific target clients. In a similar vein, Xu et al. [**?**] recompile the Linux Kernel to minimize the exposure time of persistent memory objects, thereby increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed into a single binary, known as a multivariant binary [**?**]. Each multivariant binary is randomly deployed to a client. Upon at the client, the multivariant binary executes its embedded program variants at runtime. These embedded variants can either execute in parallel to check for inconsistencies, or as a single program to randomize execution paths [**?**]. Bruschi and colleagues extend the concept of executing two variants in parallel, introducing non-overlapping and randomized memory layouts [**?**]. At the same time, Salamat et al. modifies a standard library to generate 32-bit Intel variants. These variants have a stack that grows in the opposite direction, allowing for the detection of memory inconsistencies [**?**]. Davi and colleagues propose Isomeron, an approach for execution-path randomization [**?**]. Isomeron operates by simultaneously loading the original program and a variant. It then uses a coin flip to determine which copy of the program to execute next at the function call level. Previous works have highlighted the benefits of limiting execution to only two variants in a multivariant environment. Agosta and colleagues, as well as Crane and colleagues, used more than two generated programs in the multivariant composition, thereby randomizing software control flow at runtime [**?** **?**]. Both strategies have proven effective in enhancing security by addressing known vulnerabilities, such as Just-In-Time Return-Oriented Programming (JIT-ROP) attacks [**?**] and power side-channel attacks [**?**]. Lastly, only Voulimeneas et al. [**?**] have recently proposed a multivariant execution system that enhances security by parallelizing the execution of variants across different machines.

Defensive Diversification: Lundquist and colleagues [**?**] separate the usages of Software diversification into two categories: Defensive Software Diversification and Offensive Software Diversification. Defensive Software Diversification is the traditional application of previously discussed techniques aimed at enhancing the security and reliability of software systems. The core idea is to make it more difficult for attackers to predict the system's behavior and exploit program

vulnerabilities. By doing so, even if an attacker manages to compromise one variant, the others may remain secure, thereby limiting the potential impact of the attack. Defensive diversification is often complementary with other security measures, such as encryption and intrusion detection systems, to create a multi-layered defense strategy [**?**].

Offensive Diversification: Offensive Software Diversification is a somewhat unconventional approach that uses the principles of software diversification, typically aimed at enhancing security. Yet, in the offensive context, diversification techniques may be applied to malware or other malicious code to evade detection by security software [**?**]. For example, in the context of Wasm, the seminal work of Romano et al. [**?**] proposed to intermix JavaScript and Wasm code to obfuscate JavaScript malware and make it more difficult to analyze for malware detectors. The obfuscated version of the code is functionally equivalent to the original, being, in practice, Offensive Software Diversification. Notice that, this method also measures the resilience and accuracy of security systems.

2.2.3 Open challenges

As outlined in Section 2.1.6, our primary motivation for the contributions of this thesis is the open issues within the WebAssembly ecosystem. We see potential in employing Software Diversification to address them. Based on our previous discussion, we highlight several open challenges in the realm of Software Diversification for WebAssembly. First, Wasm being an emerging technology, is still in the process of implementing defensive measures [**?**]. The process of officially adopting a new defensive measure is inherently slow, making software diversification a potentially valuable preemptive strategy. Second, despite the abundance of related work on software diversity, its exploration in the context of Wasm remains limited. Third, both defensive and offensive software diversification have been largely unexplored. Notably, the works on malware detection discussed in Section 2.1.5 suggest that offensive diversification could be useful in measuring the resilience and accuracy of security systems for WebAssembly.

■ Conclusions

In this chapter, we presented an overview of the Wasm language. This included its binary format, runtime execution concepts, and security issues. Related work was also discussed. The goal of this chapter is to establish a foundation for studying automatic diversification in Wasm. We emphasized the fact that Wasm has not been extensively researched in the field of Artificial Software Diversification. Existing implementations for Software Diversification cannot be directly applied to Wasm. Current security limitations and the absence of software diversity

approaches for Wasm inspire our work. In Chapter 3, we elaborate on the technical details that guide our contributions.

3
AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

The process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components: a frontend that converts the source code into an intermediate representation, an optimizer/transformer that modifies this representation usually for performance, and a backend that compiles the final WebAssembly binary. This architecture is illustrated in the left most part of Figure 3.1.

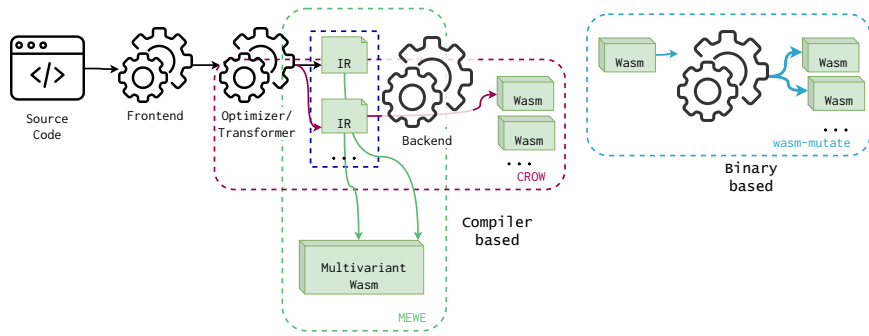


Figure 3.1: *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

[0]Comp. time 2023/10/17 16:29:53

Software Diversification, a preemptive security measure, can be integrated at various stages of this compilation process. However, applying diversification at the front-end has its limitations, as it would need a unique diversification mechanism for each language compatible with the frontend component. Conversely, diversification at later compiler stages, such as the optimizer or backend, offers a more practical alternative. This makes the latter stages of the compilers an ideal point for introducing practical Wasm diversification techniques. Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the optimizer/transformer and backend stages. This optimization/transformer component generates variants in the intermediate representation of a compiler, thereby creating artificial software diversity for WebAssembly. The variants are then compiled into WebAssembly binaries by the backend component of the compiler. Specifically, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE, which packages these variants to enable multivariant execution [**?**]. Alternatively, diversification can be directly applied to the WebAssembly binary, offering a language and compiler-agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

This dissertation contributes to the field of Software Diversification for WebAssembly by presenting two primary strategies: compiler-based and binary-based. Within this chapter, we introduce three technical contributions: CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility of our main takeaways.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combination*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combination* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

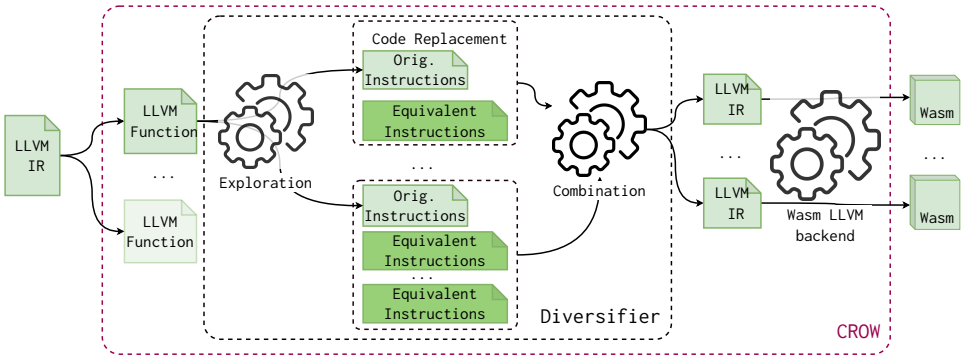


Figure 3.2: *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [**?**].*

3.1.1 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [**?**]. The search space for generating variants is delineated through an enumerative synthesis process, which systematically produces all possible code replacements for each instruction and its data flow graph in the original program. If a code replacement is identified to perform identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

Concretely, CROW is developed by extending the enumerative synthesis implementation found in Souper [**?**], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation language for each DFG. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

Notice that the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm backend, effectively reducing the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end.

CROW is carefully designed to boost the generation of variants as much as possible. First, we disable the majority of the pruning strategies. Instead of preventing the generation of commutative operations during the searching, CROW still uses such transformation as a strategy to generate program variants. Second, CROW applies code transformations independently. For instance, if a suitable replacement is identified that can be applied at N different locations in the original program, CROW will generate 2^N distinct program variants, i.e., the power set of applying the transformation or not to each location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter.

The last stage at CROW involves a custom Wasm LLVM backend, which generates the Wasm programs. For it, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.2 Constant inferring

CROW inherently introduces a novel transformation strategy called *constant inferring*, which significantly expands the variety of WebAssembly program variants. Specifically, CROW identifies segments of code that can be simplified into a single constant assignment, with a particular focus on variables that control branching logic. After applying this *constant inferring* technique, the resulting program diverges substantially from the original program structure. This is crucial for diversification efforts, as one of the primary objectives is to create variants that are as distinct as possible from the original source code [**?**]. In essence, the more divergent the variant, the more challenging it becomes to trace it back to its original form.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) %
            ↪ 1000000 != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n",
            ↪ current);
8       return 0 ;
9   }
```

Listing 3.1: *Babbage problem. Taken from [**?**].*

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current)
        ↪ ;
    return 0 ;
}
```

Listing 3.2: *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [**?**].*

CROW deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached [1]. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[2]

3.1.3 Exemplifying CROW

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

Listing 3.3: *C function that calculates the quantity $2x + x$.*

[1] In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large.

[2] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}
define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

Listing 3.6: *Wasm program variants generated from program Listing 3.3.*

Contribution paper and artifact

CROW is a compiler-based approach. It leverages enumerative synthesis to generate functionally equivalent code replacements and assembles them into diverse Wasm program variants. CROW uses SMT solvers to guarantee functional equivalence.
CROW is fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of Measurements, Attacks, and Defenses for the Web (MADWeb), NDSS 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [**?**]. Execution path randomization is a technique that randomizes the execution path of a program at runtime, i.e. at each invocation of a function, a different variant is executed [**?**]. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary called *entrypoint tampering*. The tampering is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

3.2.1 Multivariant call graph

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[**?**]. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

3.2.2 Exemplifying a Multivariant binary

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

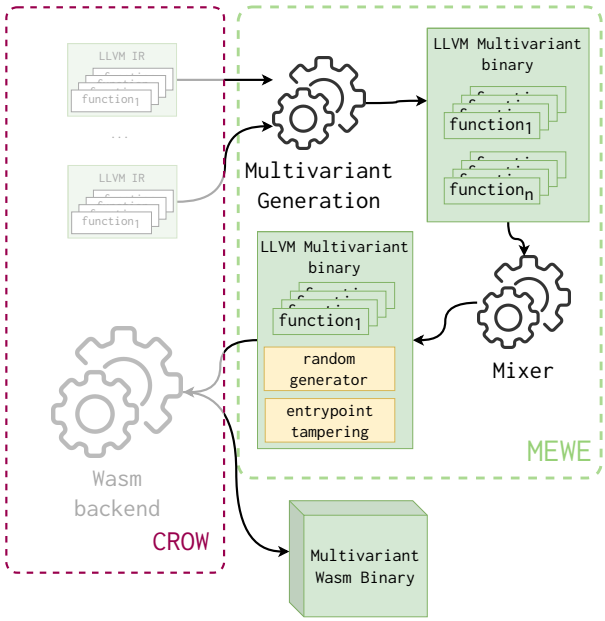


Figure 3.3: *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?**].*

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [**?**] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [**?**]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect

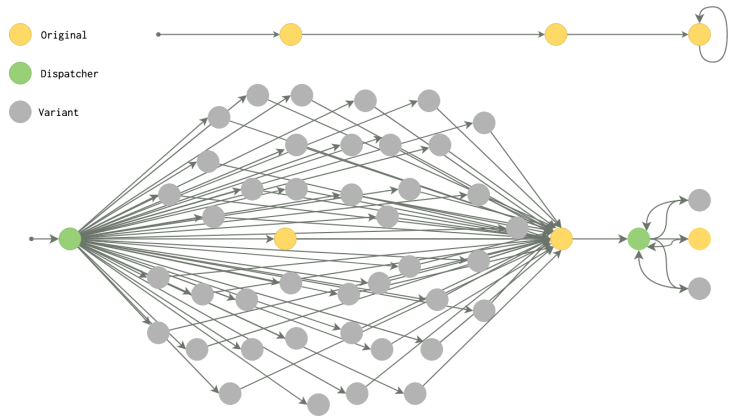


Figure 3.4: *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].*

calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

```
2  ; Multivariant foo wrapping ;
3  define internal i32 @foo(i32 %0) {
4      entry:
5        ; It first calls the dispatcher to discriminate between the created
             variants ;
6        %1 = call i32 @discriminate(i32 3)
7        switch i32 %1, label %end [
8          i32 0, label %case_43_
9          i32 1, label %case_44_
10       ]
11     ;One case for each generated variant of foo ;
12     case_43_:
13       %2 = call i32 @foo_43_(%0)
14       ret i32 %2
15     case_44_:
16       ; MEWE can inline the body of the a function variant ;
17       %3 = <body of foo_44_ inlined>
18       ret i32 %3
19     end:
20       ; The original is also included ;
21       %4 = call i32 @foo_original(%0)
22       ret i32 %4
23 }
```

Listing 3.7: *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented for the sake of understanding.*

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. Notice that, the dispatcher function is constructed using the same signature as the original function. It first calls the random generator, which returns a value used to invoke a specific function variant (see line 6 in Listing 3.7). We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [**?**] (see lines 12 to 19 in Listing 3.7), i.e., the choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. In addition, MEWE can inline function variants inside the dispatcher instead of defining them again (see line 16 in Listing 3.7). Remarkably, we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Contribution paper and artifact

MEWE provides dynamic execution path randomization by packaging variants generated out of CROW.
MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of Moving Target Defense, 2022, ACM* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [**?**], a tool that generates thousands of functionally equivalent variants out from a WebAssembly binary input. Leveraging rewriting rules and e-graphs [**?**] for software diversification, WASM-MUTATE synthesizes program variants by transforming parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

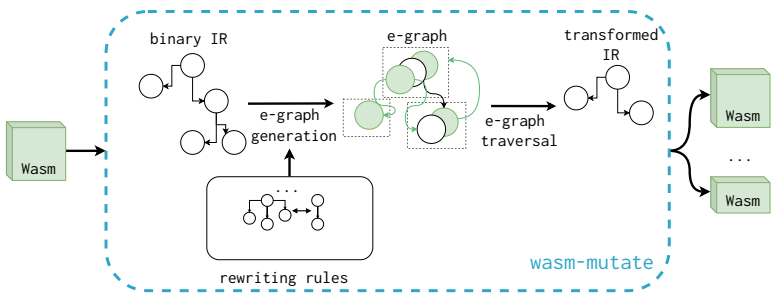


Figure 3.5: *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. The assurance of functional equivalence is rooted in the inherent

properties of the individual rewrite rules employed. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary.

WASM-MUTATE applies one transformation at a time. Notice that, the output of one applied transformation can be chained again as an input WebAssembly binary, enabling the generation of many variants, leading us to enunciate the notion of *Stacked transformation*

Definition 2. *Stacked transformation: Given an original input WebAssembly binary I and a diversifier D, stacked transformations are defined as the application of D over the binary I multiple times, i.e.,* $D(D(D(...(I))))$. *Notice that, the number of stacked transformations are the number of times the diversifier D is applied.*

3.3.1 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. In this context, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of binary targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [3].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program.

In the following example, we demonstrate a rewriting rule within the `Peephole` meta-rule that utilizes a custom `rand` operator to expand statically declared constants within any WebAssembly program function body. The unfold rewriting rule, as the name suggests, transforms statically declared constants into the sum

[3]For an in-depth explanation of the remaining meta-rules, refer to [**?**].

of two random numbers. During the generation of the WebAssembly variant, the custom `rand` operator is substituted with a randomly chosen static constant. Notice that the condition specified in the last part of the rewriting rule ensures that this predicate is satisfied.

```
LHS i32.const x

---------------------------------------------------------------

RHS (i32.add (i32.rand i32.const y))

Cond y = x - i32.rand
```

Although this rewriting approach may appear simplistic, especially because compilers often eliminate it through *Constant Folding* optimization [**?**], it stresses on the spill/reload component of the compiler when the WebAssembly binary is transpiled to machine code. Spill/reloads occur when the compiler runs out of physical registers to store intermediate calculations, resorting to specific memory locations for temporary storage. The unfold rewriting rule indirectly stresses this segment of memory. Notably, with this specific rewriting rule, we have found a CVE in the wasmtime standalone engine [**?**].

3.3.2 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [**?**]. In the context of WASM-MUTATE, e-graphs are constructed from the input WebAssembly program and the implemented rewriting rules (we detail the e-graph construction process in Section 3 of [**?**]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [**?**]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [**?**]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to fastly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the

algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

3.3.3 Exemplifying WASM-MUTATE

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule, grants the ability to append a `nop` instruction to any subexpression, a well-known low-level diversification strategy [**?**]. Conversely, the latter rule adds zero to any numeric value.

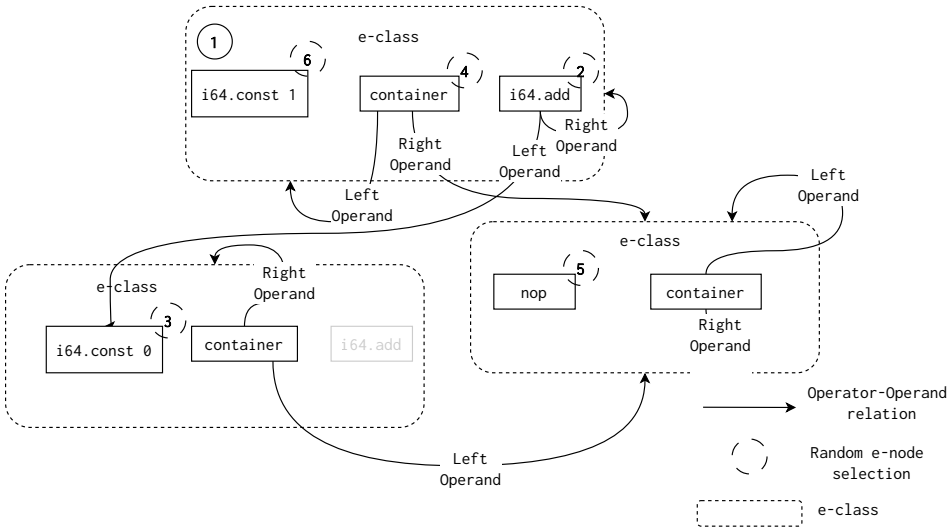


Figure 3.6: *e-graph built for rewriting the first instruction of Listing 3.8.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: *Wasm function.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: *Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.*

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr`

`= i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE uses hand-made rewriting rules and random traversals over e-graphs to provide a binary-based solution for WebAssembly diversification.
WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review at Computers & Security* `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the Bytecode Alliance organization [a]. The Bytecode Alliance is dedicated to creating secure new software foundations, building on standards such as WebAssembly and WASI.

[a] `https://bytecodealliance.org/`

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. The *Variant strength* accounts for the capability of each tool on generating variants that are preserved after the JIT

compilation of V8 and wasmtime in average. For example, a higher value of the *Variant strength* indicates that the generated variants are not reversed by JIT compilers, ensuring that the diversification is preserved in an end-to-end scenario of a WebAssembly program, i.e. from the source code to its final execution. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functionallity verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code. WASM-MUTATE, on the other hand, is a binary-based tool. It uses a set of rewriting rules and the input Wasm binary to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We have observed several interesting phenomena when aggregating the empirical data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [**? ? ?**]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because of three reasons. First, CROW and MEWE rely on SMT solvers to prove functionally equivalence, placing a bottleneck when generating variants. Second, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Section 3.1.3. Third, WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections.

On the other hand, CROW and MEWE, by using enumerative synthesis, ensure that the generated variants are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE. High preservation is especially important where the preservation of the diversification is crucial, e.g. to hinder reverse engineering.

Tool	**Input**	**Core**	**Variants in 1h**	**Target**	**Variants Strength**	**Security applications**
CROW	Source code or LLVM Ir	Enumerative synthesis with functional equivalence proved through SMT solvers	> 1k	Code section	**96%**	Hinders Static analysis reverse engineering.
MEWE	Source code or LL VM Ir	CROW, Multivariant execution	> 1k	Code and Function sections	**96%**	Hinders, static and dynamic analysis reverse engineering and, web timing-based attacks.
WASM-MUTATE	**Wasm binary**	hand-made rewriting rules, e-graph random traversals	**> 10k**	**Any Wasm section**	76%	Hinders signature-based identification, and cache timing side-channel attacks.

Table 3.1: *Comparing CROW, MEWE and WASM-MUTATE. The table columns are: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

Takeaway

Our three technical contributions serve as complementary tools that can be combined. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial.

3.4.1 Security applications

The final column of Table 3.1 emphasizes the security benefits derived from the variants produced by our three key technical contributions. One immediate advantage of altering the structure of WebAssembly binaries across different variants is the mitigation of signature-based identification, thereby enhancing resistance to static reverse engineering. Additionally, our tools generate a diverse array of code variants that are highly preserved. This implies that these variants, each with their unique WebAssembly code, retain their distinct characteristics even after being translated into machine code by JIT compilers. This high level of preservation significantly mitigates the risks associated with side-channel attacks that target specific machine code instructions, such as port contention attacks [**?**]. For instance, if a WebAssembly binary is transformed in such a manner that its resulting machine code instructions differ from the original, it becomes more challenging for a side-channel attack. Conversely, if the compiler translates the variant into machine code that closely resembles the original, the side-channel attack could still exploit those instructions to extract information about the original WebAssembly binary.

Altering the layout of a WebAssembly program inherently influences its managed memory during runtime (see **??**). This phenomenon is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential cache timing side-channels [**?**].

Last but not least, our technical contributions enhance security against web timing-based attacks [**?**] by creating variants that exhibit a wide range of execution times, including faster variants compared to the original program. This strategy is especially prominent in MEWE's approach, which develops

multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [**?**]. Adding another layer benefit from MEWE, the integration of diverse variants into multivariants can potentially disrupt dynamic reverse engineering tools such as symbolic executors [**?**]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

Takeaway

CROW, MEWE and WASM-MUTATE generate WebAssembly variants that can be used to enhance security. Overall, they generate variants that are suitable for hardening static and dynamic analysis, side-channel attacks, and, to thwart signature-based identification.

Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, proposing a novel e-graph traversal algorithm to fast spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In Chapter 4, we present two use cases that support the exploitation of these tools. Chapter 4 serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

4
EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

In this chapter we instantiate the usage of Software Diversification for offensive and defensive purposes. We present two selected use cases that exploit Software Diversification through our technical contributions presented in Chapter 3. The selected cases are representative of applications of Software Diversification for WebAssembly in browsers and standalone engines.

4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [**?**]. This is due to the essence of cryptojacking, the faster the mining, the better. Although the research of Lehmann and colleagues [**?**] suggests a decline in browser-based cryptominers, mainly due to the shutdown of Coinhive, a 2022 report by Kaspersky indicates that the use of cryptominers is on the rise [**?**]. This underscores the ongoing need for effective automatic detection of cryptojacking malware.

Let us illustrate how a malicious Wasm binary could be involved into browser cryptojacking. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

For the previous triad to work, the following steps are executed. First, the victim visits a web page infected with the cryptojacking code. The web page

[0] Comp. time 2023/10/17 16:29:53

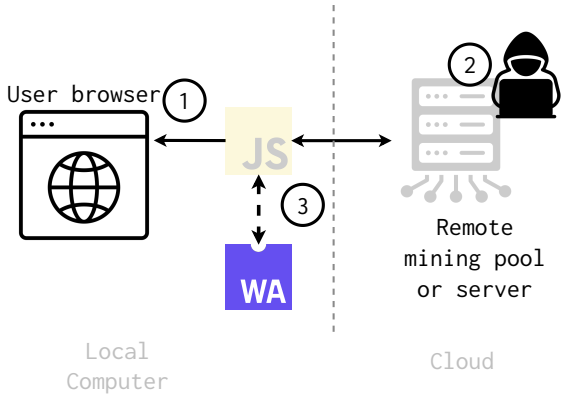


Figure 4.1: *A remote mining pool server, a JavaScript wrapper and the WebAssembly binary form the triad of a cryptojacking attack in browser clients.*

establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [**?**]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Section 2.1.5. However, malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [**?**], highlighting the critical importance of this use case. For an in-depth discussion on this topic, we direct the reader to our contribution [**?**]. The use of case illustrated in the subsequent sections uses Offensive Software Diversification for the sake of evading malware detection in WebAssembly.

TODO Rename: security model, defense model. We attack cryptojacking detection. **TODO** Do not write "Taken from"

4.1.1 Threat model: cryptojacking defense evasion

Considering the previous scenario, several techniques, as outlined in Section 2.1.5, can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. Such defense scenario is illustrated in Figure 4.2, where the WebAssembly malicious binary is blocked in ③. The primary aim of our use of case is to investigate the effectiveness of code

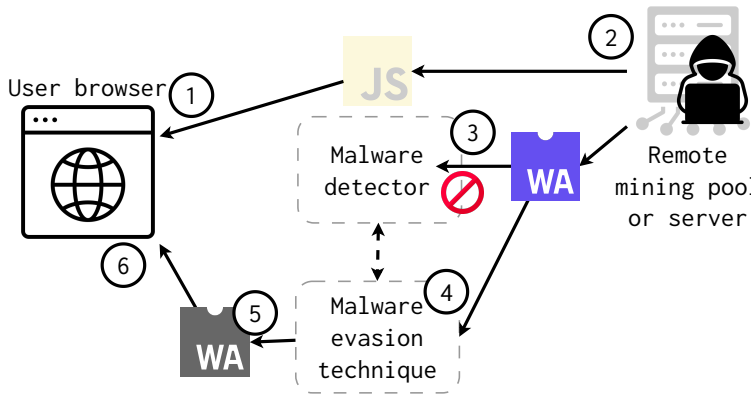


Figure 4.2: *Cryptojacking scenario in which the malware detection mechanism is bypassed by using an evasion technique.*

diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user lands on a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ② in Figure 4.2.
2. A malware detection mechanism(malware oracle) identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.
3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.
4. The attacker delivers the modified binary instead of the original one ⑤, which initiates the cryptojacking process and compromises the browser ⑥. The detection method is completely oblivious to the malicious nature of the binary, and the attack is successful.

4.1.2 Methodology

In this study, we categorize malware detection mechanisms as malware oracles, which can be of two types: binary and numeric. A binary oracle provides a binary decision, labeling a WebAssembly binary as either malicious or benign. In contrast, a numeric oracle returns a numerical value representing the confidence level of the detection.

Definition 3. *Malware oracle A malware oracle is a detection mechanism that returns either a binary decision or a numerical value indicating the confidence level of the detection.*

For empirical validation, we employ VirusTotal as a numeric oracle and MINOS [**?**] as a binary oracle. VirusTotal is an online service that analyzes files and returns a confidence score in the form of the number of antivirus that flag the input file as malware, thus qualifying as a numeric oracle. MINOS, on the other hand, converts WebAssembly binaries into grayscale images and employs a convolutional neural network for classification. It returns a binary decision, making it a binary oracle.

We use the wasmbench dataset [**?**] to establish a ground truth. After running the wasmbench dataset through VirusTotal and MINOS, we identify 33 binaries flagged as malicious by at least one VirusTotal vendor and also detected by MINOS. Then, to simulate the evasion scenario, we use WASM-MUTATE to generate WebAssembly binary variants to evade malware detection. We use WASM-MUTATE in two configurations: controlled and uncontrolled diversification.

Definition 4. *Controlled Diversification: In controlled diversification, the transformation process of a WebAssembly program is guided by a numeric oracle, which influences the probability of each transformation. For instance, WASM-MUTATE can be configured to apply transformations that minimize the oracle's confidence score. Note that controlled diversification needs a numeric oracle.*

Definition 5. *Uncontrolled Diversification: Unlike controlled diversification, uncontrolled diversification is a stochastic process where each transformation has an equal likelihood of being applied to the input WebAssembly binary.*

Based on the two types of malware oracles and diversification configurations, we examine three scenarios: 1) VirusTotal with a controlled diversification, 2) VirusTotal with an uncontrolled diversification, and 3) MINOS with an uncontrolled diversification. Notice that, the fourth scenario with MINOS and a controlled diversification is not feasible, as MINOS is a binary oracle and cannot provide the numerical values required for controlled diversification.

Our evaluation focuses on two key metrics: the success rate of evading detection mechanisms in VirusTotal and MINOS across the 33 flagged binaries, and the performance impact on the variants that successfully evade detection. The first metric measures the efficacy of WASM-MUTATE in bypassing malware detection systems. For each flagged binary, we input it into WASM-MUTATE, configured with the selected oracle and diversification strategy. We then iteratively apply transformations to the output from the preceding step. This iterative process is halted either when the binary is no longer flagged by the oracle or when a maximum of 1000 stacked transformations have been applied (see Definition 2). This process is repeated with 10 random seeds per binary

to simulate 10 different evasion experiments per binary. The second metric is crucial for validating the real-world applicability of WASM-MUTATE in evading malware detection. Specifically, if the evasion process significantly degrades the performance of the resulting binary compared to its original version, it becomes less likely to be employed in practical scenarios, such as cryptojacking. For this, we execute, end-to-end, the variants that fully evade VirusTotal when generated with WASM-MUTATE in controlled and uncontrolled diversification configurations for which we could completely reproduce the three components in Figure 4.1.

4.1.3 Results

In Table 4.1, we present a comprehensive summary of the evasion experiments presented in [**?**], focusing on two oracles: VirusTotal and MINOS[**?**]. The table is organized into two main categories to separate the results for each malware oracle. For VirusTotal, we further subdivide the results based on the two diversification configurations we employ: uncontrolled and controlled diversification. In these subsections, we provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which the uncontrolled diversification or controlled diversification setups are better than each other, the lower, the better. The MINOS section simply includes a column that specifies the number of transformations needed for complete evasion. The table has 33 + 1 rows, each representing a unique Wasm malware study subject. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.

Uncontrolled diversification to evade VirusTotal: We run uncontrolled diversification with WASM-MUTATE with a limit of 1000 iterations per binary. At each iteration, we query VirusTotal to check if the new binary evades the detection. This process is repeated with 10 random seeds per binary to simulate 10 different evasion experiments per binary. As shown in the uncontrolled diversification part of Table 4.1, we successfully generate variants that evade detection for 30 out of 33 binaries. The mean value of iterations needed to generate a variant that evades all detectors ranges from 120 to 635 stacked transformations. The mean number of iterations needed is always less than 1000 stacked transformations. There are 3 binaries for which the uncontrolled diversification setup does not completely evade the detection. In these three cases, the algorithm misses 5 out 31, 6 out of 30 and 5 out 26 detectors. The explanation is the maximum number of iterations 1000 we use for our experiments. However, having more iterations seems not a realistic scenario. For example, if some transformations increment the binary size during the transformation, a

	VirusTotal					MINOS[?]
Hash	#D	Uncontrolled diversification		Controlled diversification		
		Max. evaded	Mean trans.	Max. evaded	Mean trans.	Mean trans.
47d29959	31	**26**	N/A	19	N/A	100
9d30e7f0	30	**24**	N/A	17	N/A	419
8ebf4e44	26	**21**	N/A	13	N/A	92
c11d82d	20	20	**355**	20	446	115
0d996462	19	19	**401**	19	697	24
a32a6f4b	18	18	635	18	**625**	1
fbdd1efa	18	18	**310**	18	726	1
d2141ff2	9	9	**461**	9	781	81
aafff587	6	6	484	6	**331**	1
046dc081	6	6	404	6	**159**	33
643116ff	6	6	**144**	6	436	47
15b86a25	4	4	253	4	**131**	1
006b2fb6	4	4	**282**	4	380	1
942be4f7	4	4	200	4	200	29
7c36f462	4	4	236	4	**221**	85
fb15929f	4	4	**297**	4	475	1
24aae13a	4	4	**252**	4	401	980
000415b2	3	3	302	3	**34**	960
4cbdbbb1	3	3	295	3	**72**	1
65debcbe	2	2	131	2	**33**	38
59955b4c	2	2	130	2	**33**	38
89a3645c	2	2	431	2	**107**	108
a74a7cb8	2	2	124	2	**33**	38
119c53eb	2	2	104	2	**18**	1
089dd312	2	2	153	2	**123**	68
c1be4071	2	2	130	2	**33**	38
dceaf65b	2	2	140	2	132	66
6b8c7899	2	2	143	2	**33**	38
a27b45ef	2	2	145	2	**33**	33
68ca7c0e	2	2	137	2	**33**	38
f0b24409	2	2	127	2	**11**	33
5bc53343	2	2	118	2	**33**	33
e09c32c5	1	1	**120**	1	488	15
Median			218		131	38

Table 4.1: *The table has two main categories for each malware oracle, corresponding to the two oracles we use: VirusTotal and MINOS. For VirusTotal, divide the results based on the two diversification configurations: uncontrolled and controlled diversification. We provide columns that indicate the number of VirusTotal vendors that flag the original binary as malware (#D), the maximum number of successfully evaded detectors (Max. #evaded), and the average number of transformations required (Mean #trans.) for each sample. We highlight in bold text the values for which diversification setups are better than each other, the lower, the better. The MINOS section includes a column that specifies the number of transformations needed for complete evasion. The final row offers the median number of transformations required for evasion across our evaluated setups and oracles.*

considerably large binary might be impractical for bandwidth reasons. Overall, uncontrolled diversification with WASM-MUTATE clearly decreases the detection rate by VirusTotal antivirus vendors for cryptojacking malware, achieving total evasion of WebAssembly cryptojacking malware in 30/33 (90%) of the malware dataset.

Takeaway

When compared with the data in Table 3.1, WASM-MUTATE generates an average of nearly 10000 variants per binary within an hour. Thus, WASM-MUTATE is capable of successfully evading detection systems in just a matter of minutes.

Controlled diversification to evade VirusTotal: Uncontrolled diversification does not guide the diversification based on the number of evaded detectors, it is purely random, and has some drawbacks. For example, some transformations might suppress other transformations previously applied. We have observed that, by carefully selecting the order and type of transformations applied, it is possible to evade detection systems in fewer iterations. This can be appreciated in the results of the controlled diversification part of Table 4.1. Analyzing the data in Table 4.1, we observe that the controlled diversification setup successfully generates variants that totally evade the detection for 30 out of 33 binaries, it thus as good as the uncontrolled setup. The iterations needed for the controlled diversification setup are 92% of the needed on average for the uncontrolled diversification setup. For 21 of 30 binaries that evade detection entirely, we observe that the mean number of oracle calls needed is lower than those in the baseline evasion algorithm. For example, `f0b24409` needs 11 oracle calls with controlled diversification setup to fully evade VirusTotal, while for the uncontrolled one, it needs 127 oracles calls. For those 21 binaries, it needs only 40% of the calls the controlled diversification setup needs.

Uncontrolled diversification to evade MINOS: Additionally, relying solely on VirusTotal as the detection mechanism could be problematic, especially when specialized solutions exist exclusively for WebAssembly, unlike the general-purpose vendors in VirusTotal. To address this concernt, we also assessed the efficacy of our evasion algorithms against MINOS, a WebAssembly-specific antivirus. We iteratively applied random mutations to the original malware binary until either MINOS was fully evaded or a maximum iteration limit was reached. This process was repeated 10 times for each binary. The outcomes are displayed in the last column of Table 4.1. The last row of Table 4.1 shows that WebAssembly diversification requires fewer iterations to evade MINOS than VirusTotal, meaning that it is easier to evade MINOS. The median number of iterations needed overall for evading VirusTotal is 218 for the uncontrolled diversification setup, and 131 for the controlled diversification setup, while for

MINOS is 38. Remarkably, WASM-MUTATE totally evades detection for 8 out of 33 binaries in one single iteration in the case of MINOS. This shows that the MINOS model is fragile wrt binary diversification.

Takeaway

According to our results, VirusTotal can be considered better than MINOS wrt to cryptojacking detection. The main reason is that a wider spectrum of antivirus vendors is used in VirusTotal, while MINOS is a single detector. Therefore, this advocates for the use of multiple malware oracles (meta-oracles) to detect cryptojacking malware in browsers, even in the case Wasm-specific detection mechanism.

Performance To evaluate the real-world efficacy of WASM-MUTATE in evading malware detection, we focused on six binaries that we could build and execute end-to-end, as these had all three components outlined in Figure 4.1. For these binaries, we replace the original WebAssembly code with variants generated using VirusTotal as the malware oracle and WASM-MUTATE for both controlled and uncontrolled diversification configurations. We then execute both the original and the generated variants. We assessed the variants based on the hash generation rate.

We have found that 19% of the generated variants outperformed the original cryptojacking binaries. This improvement is attributed to WASM-MUTATE's ability to introduce code optimizations. Additionally, debloating transformations, which eliminate unnecessary structures and dead code, resulted in a higher hash generation rate during the initial seconds of mining, likely due to faster compilation times. This suggests that focused optimization serves as a valuable tool for evasion in browsers. On the contrary, 80% of the generated variants are less efficient than the original binary, with the least efficient variant operating at only 20% of the original hash generation rate. This performance drop is primarily due to non-optimal transformations introduced by WASM-MUTATE. Variants generated through uncontrolled diversification are generally slower. In summary, controlled diversification yielded variants that evaded VirusTotal detection with minimal performance overhead—the worst-performing variant was only 1.93 times slower than the original.

Contribution paper

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of our work are actionable, as we also provide quantitative evidence on specific malware transformations on which detection methods can focus. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

4.2 Defensive Diversification: Speculative Side-channel protection

As discussed in **??**, WebAssembly is quickly becoming a cornerstone technology in backend systems. Leading companies like Cloudflare and Fastly are championing the integration of WebAssembly into their edge computing platforms, thereby enabling developers to deploy applications that are both modular and securely sandboxed. These client-side WebAssembly applications are generally architected as isolated, single-responsibility services, a model referred to as Function-as-a-Service (FaaS) [**?** **?**]. The operational flow of WebAssembly binaries in FaaS platforms is illustrated in Figure 4.3.

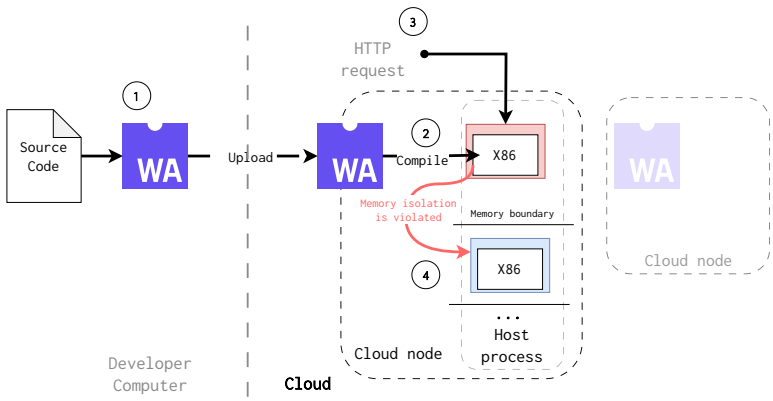


Figure 4.3: *WebAssembly binaries on FaaS platforms. Developers can submit any WebAssembly binary to the platform to be executed as a service in a sandboxed and isolated manner. Yet, WebAssembly binaries are not immune to Spectre attacks.*

The fundamental advantage of using WebAssembly in FaaS platforms lies in its ability to encapsulate thousands of client WebAssembly binaries within

a singular host process. A developer could compile its source code into a WebAssembly program suitable for the cloud platform and then submit it (① in Figure 4.3). This host process is then disseminated across a network of servers and data centers (② in Figure 4.3). These platforms convert WebAssembly programs into native code, which is subsequently executed in a sandboxed environment. Host processes can then instantiate new WebAssembly sandboxes for each client function, executing them in response to specific user requests with nanosecond-level latency (③ in Figure 4.3). This architecture inherently isolates WebAssembly binary executions from each other as well as from the host process, enhancing security.

However, while WebAssembly is engineered with a strong on security and isolation, it is not entirely immune to vulnerabilities such as Spectre attacks [**?** **?**] (④ in Figure 4.3). In the sections that follow, we explore how software diversification techniques can be employed to fortify WebAssembly binaries against such attacks. Dale ven

For an in-depth discussion on this topic, we direct the reader to our contribution [**?**].

4.2.1 Threat model: speculative side-channel attacks

To illustrate the threat model concerning WebAssembly programs in FaaS platforms, consider the following scenarios. Developers, including potentially malicious actors, have the ability to submit any WebAssembly binary to the FaaS platform. A malicious actor could then upload a WebAssembly binary that, once compiled to native code, employs Spectre attacks to either leak sensitive information from the host process or violate Control Flow Integrity (CFI). Furthermore, even if a submitted WebAssembly binary is not intentionally malicious, it may still be vulnerable to Spectre attacks. For instance, a malicious actor could exploit this vulnerability by executing the susceptible binary through the FaaS service.

Spectre attacks exploit hardware-based prediction mechanisms to trigger mispredictions, leading to the speculative execution of specific instruction sequences that are not part of the original, sequential execution flow. By taking advantage of this speculative execution, an attacker can potentially access sensitive information stored in the memory allocated to other WebAssembly instance(including itself) or even the host process itself. This poses a significant risk, compromising both the security and integrity of the overall system.

Narayan and colleagues [**?**] have categorized potential Spectre attacks on Wasm binaries into three distinct types, each corresponding to a specific hardware predictor being exploited and a particular FaaS scenario: Branch Target Buffer Attacks, Return Stack Buffer Attacks, and Pattern History Table Attacks defined as follows:

Program	Attack
btb_breakout	Spectre branch target buffer (btb)
btb_leakage	Spectre branch target buffer(btb)
ret2spec	Spectre Return Stack Buffer (rsb)
pht	Spectre Pattern History Table (pht)

Table 4.2: *WebAssembly program name and its respective attack.*

1. The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target.

2. The Spectre Return Stack Buffer (rsb) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions.

3. The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition.

4.2.2 Methodology

Our goal is to empirically validate that Software Diversification can effectively mitigate the risks associated with Spectre attacks in WebAssembly binaries. The green-highlighted section in Figure 4.4 illustrates how Software Diversification can be integrated into the FaaS platform workflow. The core idea is to generate unique and diverse WebAssembly variants that can be randomized at the time of deployment. For this use case, we employ WASM-MUTATE as our tool for Software Diversification.

To empirically demonstrate that Software Diversification can indeed mitigate Spectre vulnerabilities, we reuse the WebAssembly attack scenarios proposed by Narayan and colleagues in their work on Swivel [**?**]. Swivel is a compiler-based strategy designed to counteract Spectre attacks on WebAssembly binaries by linearizing their control flow during machine code compilation. Our approach differs from theirs in that it is binary-based, compiler-agnostic, and platform-agnostic; we do not propose altering the deployment or toolchain of FaaS platforms. Although our experiments are conducted prior to submitting the WebAssembly binary to the FaaS platform, we argue that WebAssembly binary diversification could be implemented at any stage of the FaaS workflow. The same argument holds by using any other diversification tool presented in this dissertation (see Chapter 3).

To measure the efficacy of WASM-MUTATE in mitigating Spectre, we diversify four WebAssembly binaries proposed in the Swivel study. The names of

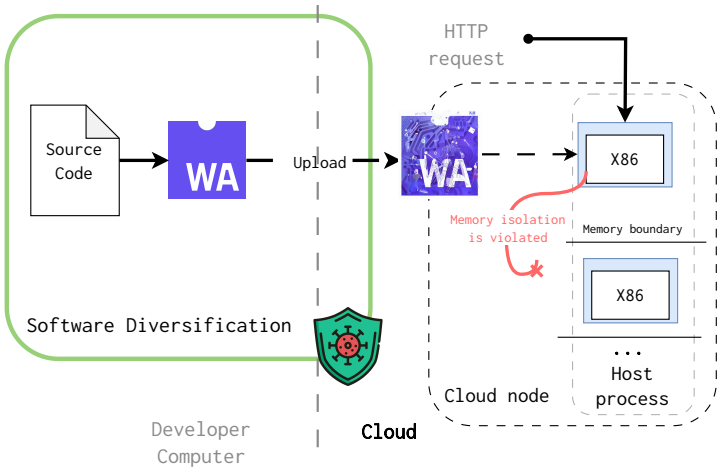


Figure 4.4: *Diversifying WebAssembly binaries to mitigate Spectre attacks in FaaS platforms.*

these programs and the specific attacks we examine are available in [**?**]. For each of these four binaries, we generate up to 1000 random stacked transformations (see Definition 2) using 100 distinct seeds, resulting in a total of 100,000 variants for each original binary. At every 100th stacked transformation for each binary and seed, we assess the impact of diversification on the Spectre attacks by measuring the attack bandwidth for data exfiltration. This metric not only captures the success or failure of the attacks but also quantifies the extent to which data exfiltration is hindered. For example, a variant that still leaks data but does so at an impractically slow rate would be considered hardened against the attack.

Definition 6. *Attack bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* T *and* $K = k_1, k_2, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

4.2.3 Results

Figure 4.5 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs: btb_breakout and btb_leakage with the btb attack. The Y-axis represents the exfiltration bandwidth (see Definition 6). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by the green violinplots.

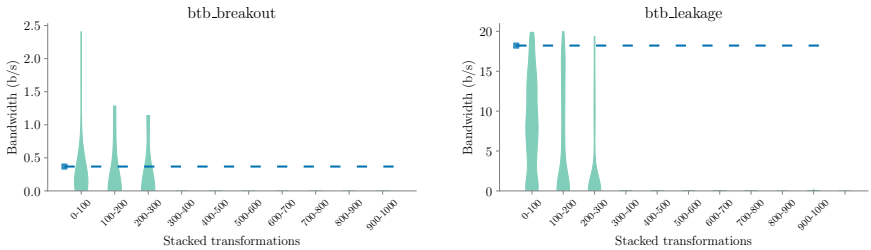


Figure 4.5: *Impact of WASM-MUTATE over btb_breakout and btb_leakage binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth. After 200 stacked transformations, the exfiltration bandwidth drops to zero.*

Population Strength: For the binaries btb_breakout and btb_leakage, WASM-MUTATE exhibits a high level of effectiveness, generating variants that leak less information than the original in 78% and 70% of instances, respectively. For both programs, after applying 200 stacked transformations, the exfiltration bandwidth drops to zero. This implies that WASM-MUTATE is capable of synthesizing variants that are entirely protected from the original attack.

Takeaway

As indicated in Table 3.1, generating a variant with 200 stacked transformations can be accomplished in just a matter of minutes. When scaled to the scope of a global FaaS platform, this means that a unique, fortified variant could be deployed for each machine and even for each fresh WebAssembly spawned per user request.

As illustrated in Figure 4.6, similarly to Figure 4.5, WASM-MUTATE significantly impacts the programs ret2spec and pht when subjected to their respective attacks. In 76% of instances for ret2spec and 71% for pht, the generated variants demonstrated reduced attack bandwidth compared to the original binaries. The plots reveal that a notable decrease in exfiltration bandwidth occurs after applying at least 100 stacked transformations. While both programs show signs of hardening through reduced attack bandwidth, this effect is not immediate and requires a substantial number of transformations to become effective. Additionally, the bandwidth distribution is more varied for these two programs compared to the two previous ones. Our analysis suggests a correlation between the reduction in attack bandwidth and the complexity of the binary being diversified. Specifically, ret2spec and pht are substantially

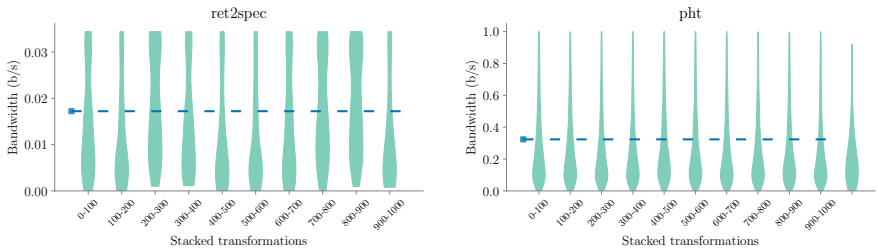


Figure 4.6: *Impact of WASM-MUTATE over ret2spec and pht binaries. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green violinplots. Overall, for both programs approximately 70% of the variants have less data leakage bandwidth.*

larger programs, containing over 300,000 instructions, compared to btb_breakout and btb_leakage, which have fewer than 800 instructions. Therefore, given that WASM-MUTATE performs incremental transformations, the probability of affecting critical components to hinder attacks decreases in larger binaries.

Managed memory impact: The success in diminishing exfiltration is explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. We have identified four primary factors responsible for the divergence in memory accesses among WASM-MUTATE generated variants. First, modifications to the binary layout—even those that don't affect executed code—inevitably alter memory accesses within the program's stack. Specifically, WASM-MUTATE generates variants that modify the return addresses of functions, which consequently leads to differences in execution flow and memory accesses. Second, one of our rewriting rules incorporates artificial global values into Wasm binaries. Since these global variables are inherently manipulated via the stack, and given that the stack is located within linear memory, their access inevitably affects the managed memory (see **??**). Third, WASM-MUTATE injects 'phantom' instructions which don't aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the wasmtime compiler, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating diverse managed memory accesses(see the example at Section 3.3.1). Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does.

Disrupting accurate timers: Cache timing side-channel attacks, including for the four binaries analyzed in this case study, depend on precise timers to measure cache access times. Disrupting these timers can effectively neutralize the attack. For example, in other contexts, Firefox employs a strategy to counter timing attacks by randomizing its built-in JavaScript timer [**?**]. WASM-MUTATE inherently adopts a similar approach, introducing perturbations in the timing steps of Wasm variants in case they are malicious. This is illustrated in Listing 4.1 and Listing 4.2, where the former shows the original time measurement and the latter presents a variant with WASM-MUTATE-introduced operations. WASM-MUTATE is particularly effective in disrupting cache access timers. By introducing additional instructions, the inherent randomness in the time measurement of a single or a few instructions is amplified, thereby reducing the timer's accuracy.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.1: *Wasm timer used in btb_breakout program.*

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 4.2: *Variant of btb_breakout with more instructions added in between time measurement.*

Padding speculated instructions: Additionally, CPUs have a limit on the number of instructions they can cache. WASM-MUTATE injects instructions to potentially exceed this limit, effectively disabling the speculative execution of memory accesses. This approach is akin to padding [**?**], as demonstrated in Listing 4.3 and Listing 4.4. This padding disrupts the binary code's layout in memory, hindering the attacker's ability to initiate speculative execution. Even if speculative execution occurs, the memory access does not proceed as the attacker intended.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

Listing 4.3: *Two jump locations in btb_breakout. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.*

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

Listing 4.4: *Variant of btb_breakout with more instructions added indindinctly between jump places.*

Drawbacks: We observed that the exfiltration bandwidth tends to increase in variants with only a few transformations. This suggests that not all transformations uniformly contribute to reducing data leakage. Several key factors contribute to this phenomenon. First, as emphasized previously in Section 4.1, uncontrolled diversification can be counterproductive if a specific objective, e.g., if a cost function, is not established at the beginning of the diversification process. Second, while some transformations yield distinct Wasm binaries, their compilation produces identical machine code. Transformations that are not preserved(see Section 3.4) undermine the effectiveness of diversification. For example, incorporating random `nop` operations directly into Wasm does not modify the final machine code as the `nop` operations are often removed by the compiler. The same phenomenon is observed with transformations to custom sections of WebAssembly binaries. Additionally, it is important to note that transformed code doesn't always execute, i.e., WASM-MUTATE may generate dead code.

Contribution paper

Software diversification crafts WebAssembly binaries that are resilient to Spectre-like attacks. By integrating a software diversification layer into WebAssembly binaries deployed on Function-as-a-Service (FaaS) platforms, security can be significantly bolstered. This approach allows for the deployment of unique and diversified WebAssembly binaries, potentially utilizing a distinct variant for each cloud node, thereby enhancing the overall security. The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

Conclusions

In this chapter, we discuss two sides of Software Diversification as applied to WebAssembly: Offensive Software Diversification and Defensive Software Diversification. The term *Offensive Software Diversification* may seem counterintuitive at first glance, but its role is to underscore both the capabilities and the latent security risks inherent in applying Software Diversification to WebAssembly. Our research indicates that there are ways for enhancing the automated detection of cryptojacking malware in WebAssembly, e.g. by testing their resilience with WebAssembly malware variants. On the other hand, Defensive Software Diversification acts as a preemptive safeguard, specifically to mitigate the risks posed by Spectre attacks. In the subsequent chapter, we will consolidate the principal conclusions of this dissertation and describe directions for future research.

5 CONCLUSIONS AND FUTURE WORK

5.1 Summary of technical contributions

5.2 Summary of empirical findings

5.3 Future Work

TODO WASM-MUTATE slicing

We have observed that some transformations can be applied in any order. This means that different sequences of transformations can produce the same binary variant. This often happens when two mutation targets inside the binary are different, such as two disjoint pieces of code. Therefore, a potential parallelization for the baseline algorithm is possible as soon as transformation sequences do not interfere with others.

To further enhance the detection capabilities of MINOS, we believe in binary canonicalization [**?**]. By creating a canonical representation of the malware variant before training and inference, one would help the classifier to better generalize. This is feasible as it is a preprocessing step in the pipeline. We believe this is an interesting direction for future work.

Furthermore, WASM-MUTATE can benefit from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could incorporate the transformations generated by these tools as rewriting rules.

Moreover, the WebAssembly ecosystem is still in its infancy compared to more mature programming environments. A 2021 study by Hilbig et al. found only 8,000 unique WebAssembly binaries globally[**?**], a fraction of the 1.5 million and 1.7 million packages available in npm and PyPI, respectively. This limited dataset poses challenges for machine learning-based analysis tools, which require extensive data for effective training. The scarcity of WebAssembly programs

[0] Comp. time 2023/10/17 16:29:53

also exacerbates the problem of software monoculture, increasing the risk of compromised WebAssembly programs being consumed[**?**]. This dissertation aims to mitigate these issues by introducing a comprehensive suite of tools designed to enhance WebAssembly security through Software Diversification and to improve testing rigor within the ecosystem.

Program Normalization WASM-MUTATE was previously employed successfully for the evasion of malware detection, as outlined in [**?**]. The proposed mitigation in the prior study involved code normalization as a means of reducing the spectrum of malware variants. Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be compiled with wasmtime, effectively eliminating approx. 25% of malware variants according to our preservation statistics for wasmtime. This approach could substantially enhance the efficiency and precision of malware detection systems.

Fuzzing WebAssembly compilers with WASM-MUTATE In fuzzing campaigns, generating well-formed inputs is a significant challenge [**?**]. This is particularly true for fuzzing compilers, where the inputs should be executable yet intricate enough programs to probe various compiler components. WASM-MUTATE could address this challenge by generating semantically equivalent variants from an original Wasm binary, enhancing the scope and efficiency of the fuzzing process. A practical example of this occurred in 2021, when this approach led to the discovery of a wasmtime security CVE [**?**]. Through the creation of semantically equivalent variants, the spill/reload component of cranelift was stressed, resulting in the discovery of the before-mentioned CVE.

Mitigating Port Contention with WASM-MUTATE Rokicki et al. [**?**] showed the practicality of a covert side-channel attack using port contention within WebAssembly code in the browser. This attack fundamentally relies on the precise prediction of Wasm instructions that trigger port contention. To combat this security concern, WASM-MUTATE could be conveniently implemented as a browser plugin. WASM-MUTATE has the ability to replace the Wasm instructions used as port contention predictor with other instructions. This would inevitably remove the port contention in the specific port used to conduct the attack, hardening browsers against such malicious maneuvers.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with webassembly," *PLDI*, 2017.

[2] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Commun. ACM*, vol. 61, p. 107–115, nov 2018.

[3] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[4] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*, pp. 381–389, Springer, Cham, 2018.

[5] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

[6] D. Bryant, "Webassembly outside the browser: A new foundation for pervasive computing," in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[7] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, 2021.

[8] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4, IEEE, 2020.

[9] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," *Proceedings of the Web Conference 2021*, 2021.

[10] L. Wagner, M. Mayer, A. Marino, A. Soldani Nezhad, H. Zwaan, and I. Malavolta, "On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, (New York, NY, USA), p. 72–82, Association for Computing Machinery, 2023.

[11] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[12] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *International Conference on Web Engineering*, pp. 328–336, Springer, 2021.

[13] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, p. 265–279, 2020.

[14] R. Gurdeep Singh and C. Scholliers, "Warduino: A dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, (New York, NY, USA), pp. 27–36, ACM, 2019.

[15] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *Static Analysis* (G. Singh and C. Urban, eds.), (Cham), pp. 74–103, Springer Nature Switzerland, 2022.

[16] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, 2022.

[17] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic execution for webassembly (artifact)," Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[18] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, ", : Sfi safety for native-compiled wasm," *Network and Distributed Systems Security (NDSS) Symposium*.

[19] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[20] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational verification of cryptographic implementations in webassembly," in *2021 IEEE Secure Development Conference (SecDev)*, pp. 94–102, 2021.

[21] Q. Stiévenart and C. De Roover, "Wassail: a webassembly static analysis library," in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[22] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 753–757, 2023.

[23] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[24] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv preprint arXiv:2110.15433*, 2021.

[25] Q. Stiévenart, D. Binkley, and C. De Roover, "Dynamic slicing of webassembly binaries," in *39th IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2023.

[26] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2031–2042, Association for Computing Machinery, 2022.

[27] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[28] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1433–1450, USENIX Association, Aug. 2021.

[29] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "Wave: a verifiably secure webassembly sandboxing runtime," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2940–2955, 2023.

[30] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pp. 23–42, Springer, 2019.

[31] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1714–1730, 2018.

[32] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1129–1140, 2020.

[33] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system.," in *NDSS*, 2021.

[34] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*, pp. 122–142, Springer, 2018.

[35] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 313–326, 2018.

[36] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.

[37] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A first look at code obfuscation for webassembly," in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, (New York, NY, USA), p. 140–145, Association for Computing Machinery, 2022.

[38] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 119, 2018.

[39] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 2109–2122, Association for Computing Machinery, 2018.

[40] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, (New York, NY, USA), p. 1182–1194, Association for Computing Machinery, 2022.

[41] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting c programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, (New York, NY, USA), p. 1713–1722, Association for Computing Machinery, 2022.

[42] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48, sep 2015.

[43] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*, vol. 1. Springer, 2005.

[44] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, pp. 60–66, 2010.

[45] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 124–134, Association for Computing Machinery, 2011.

[46] Avizienis and Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[47] F. B. Cohen, "Operating system protection through program evolution.," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[48] G. R. Lundquist, V. Mohan, and K. W. Hamlen, "Searching for software0 diversity: Attaining artificial diversity through program synthesis," in *Proceedings of the 2016 New Security Paradigms Workshop*, NSPW '16, (New York, NY, USA), p. 80–91, Association for Computing Machinery, 2016.

[49] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, p. 437–449, apr 1975.

[50] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pp. 67–72, 1997.

[51] T. Jackson, *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine, 2012.

[52] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Trans. Archit. Code Optim.*, vol. 8, jan 2012.

[53] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE, 2013.

[54] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[55] R. M. Tsoupidi, R. C. Lozano, and B. Baudry, "Constraint-based software diversification for efficient mitigation of code-reuse attacks," *ArXiv*, vol. abs/2007.08955, 2020.

[56] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the USENIX Security Symposium*, 2003.

[57] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the USENIX Security Symposium*, pp. 271–286, 2005.

[58] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, (New York, NY, USA), p. 16–27, Association for Computing Machinery, 1990.

[59] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity.," in *NDSS*, pp. 8–11, 2015.

[60] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, (Los Alamitos, CA, USA), pp. 1101–1116, IEEE Computer Society, may 2022.

[61] M. T. Aga and T. Austin, "Smokestack: thwarting dop attacks with runtime stack layout randomization," in *Proc. of CGO*, pp. 26–36, 2019.

[62] S. Lee, H. Kang, J. Jang, and B. B. Kang, "Savior: Thwarting stack-based memory safety violations by randomizing stack layout," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[63] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 429–438, 2006.

[64] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proc. of ASPLOS*, pp. 987–1000, 2020.

[65] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of CCS*, pp. 272–280, 2003.

[66] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proc. CCS*, pp. 281–289, 2003.

[67] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Tech. Rep. CS-02-197, Carnegie Mellon University, 2002.

[68] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *IFIP International Conference on Information Security Theory and Practice*, pp. 136–152, Springer, 2016.

[69] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," *arXiv e-prints*, p. arXiv:2308.03123, Aug. 2023.

[70] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 216–226, 2014.

[71] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 1027–1040, Association for Computing Machinery, 2019.

[72] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[73] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[74] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *SIGARCH Comput. Archit. News*, vol. 44, p. 297–310, mar 2016.

[75] M. Zalewski, "American fuzzy lop," 2017.

[76] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Information and Communications Security* (J. Lopez, S. Qing, and E. Okamoto, eds.), (Berlin, Heidelberg), pp. 187–199, Springer Berlin Heidelberg, 2004.

[77] V. Singhal, A. A. Pillai, C. Saumya, M. Kulkarni, and A. Machiry, "Cornucopia: A framework for feedback guided generation of binaries," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[78] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. of USENIX Security Symposium*, USENIX-SS'06, 2006.

[79] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *Proc. of the Int. Performance, Computing, and Communications Conference*, 2007.

[80] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Stopping buffer overflow attacks at run-time: Simultaneous multi-variant program execution on a multicore processor," tech. rep., Technical Report 07-13, School of Information and Computer Sciences, UCIrvine, 2007.

[81] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.

[82] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1320–1333, 2015.

[83] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*, pp. 77–98, Springer, 2011.

[84] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can code polymorphism limit information leakage?," in *IFIP International Workshop on Information Security Theory and Practices*, pp. 1–21, Springer, 2011.

[85] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," in *Proceedings of the 14th European Workshop on Systems Security*, pp. 41–47, 2021.

[86] I. Bow, N. Bete, F. Saqib, W. Che, C. Patel, R. Robucci, C. Chan, and J. Plusquellic, "Side-channel power resistance for encryption algorithms using implementation diversity," *Cryptography*, vol. 4, no. 2, 2020.

[87] R. L. Castro, C. Schmitt, and G. D. Rodosek, "Armed: How automatic malware modifications can evade static detection?," in *2019 5th International Conference on Information Management (ICIM)*, pp. 20–27, 2019.

[88] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[89] J. Cabrera Arteaga, "Artificial software diversification for webassembly," 2022. QC 20220909.

[90] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[91] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[92] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 403–422, Springer, 2018.

[93] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[94] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[95] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[96] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *arXiv e-prints*, p. arXiv:2309.07638, Sept. 2023.

[97] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[98] "Stop a wasm compiler bug before it becomes a problem | fastly." `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`, 2021.

[99] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[100] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), p. 264–276, Association for Computing Machinery, 2009.

[101] T. D. Morgan and J. W. Morgan, "Web timing attacks made practical," *Black Hat*, 2015.

[102] T. Schnitzler, K. Kohls, E. Bitsikas, and C. Pöpper, "Hope of delivery: Extracting user locations from mobile instant messengers," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023.

[103] Kaspersky, "The state of cryptojacking in the first three quarters of 2022," 2022.

[104] Mozilla, "Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta ," 2019.

[105] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "Webassembly diversification for malware evasion," *Computers & Security*, vol. 131, p. 103296, 2023.

[106] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, 2019.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb

https://doi.org/10.14722/madweb.2021.23004

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

https://www.sciencedirect.com/science/article/pii/S0167404823002067

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228